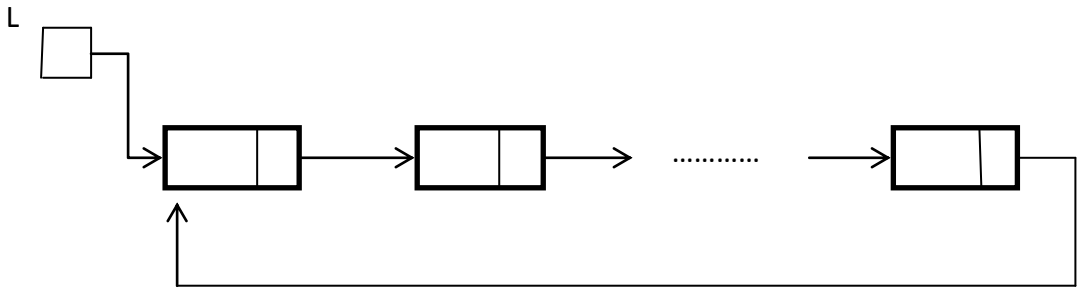
AUTRES LISTES

Après avoir étudié la liste chaînée linéaire unidirectionnelle, on va se pencher sur d'autres listes :

- Liste chaînée circulaire unidirectionnelle.
- Liste chaînée linéaire bidirectionnelle.
- Liste chaînée circulaire bidirectionnelle.

1. Liste chaînée circulaire unidirectionnelle.



On remarque que le dernier nœud est relié au premier : on dit que c'est un anneau.

- Son implémentation est exactement la même que celle d'une liste chaînée linéaire.

```
#includ <stdlib.h>
Typedef  struct node node ;
Struct node {
                Item info;
                Struct node *suiv;
              }
Typedef node* liste;
```

NB: Le seul cas où on parle de NULL, c'est quand la liste est vide!

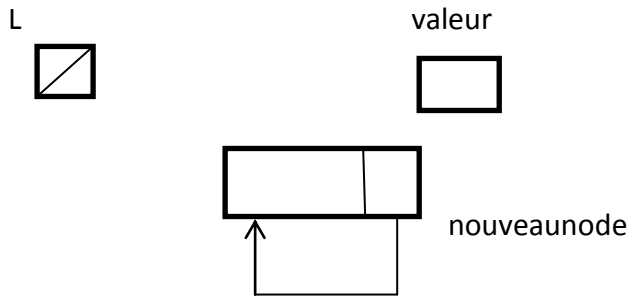
- Quelques opérations de base.
  - Le parcours d'une liste circulaire non vide.

```
Void parcoursliste(liste L)
{
   Node* Pt = L ;
   While Pt -> != L
         Pt = Pt -> suiv ;
}
```

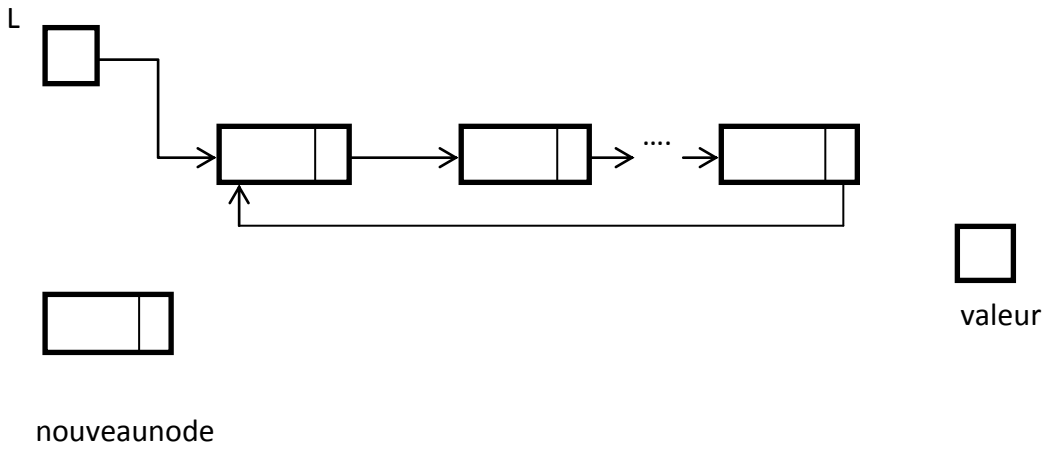
- Ajout d'une valeur en tête de liste.

```
Liste AjoutEnTete(liste L ; item valeur) ;
{
  Node* nouveaunode = malloc(sizeof (node)) ;
  Nouveaunode -> info = valeur ;
  If L == NULL
     Nouveaunode ->suiv = nouveaunode ;
  Else
      {
        Node* Pt = L ;
        While Pt -> suiv != L
              Pt = Pt -> suiv;
        Pt -> suiv = nouveaunode;
        Nouveaunode -> suiv = L;
      }
   Return nouveaunode;
}
```

Cas particulier :



Cas général :



- Ajout d'une valeur en fin de liste.

```
Liste AjoutEnFin(liste L, item valeur) ;
{
    Node* nouveaunode = malloc(sizeof (node)) ;
    Nouveaunode -> info = valeur ;
    If L == NULL
     {
        Nouveaunode ->suiv = nouveaunode ;
        Return nouveaunode ;

     }
    Else
      {
        Node* Pt = L ;
        While Pt -> suiv != L
              Pt = Pt -> suiv ;
        Pt -> suiv = nouveaunode ;
        Nouveaunode -> suiv = L ;
        Return L ;
       }
  }
```

NB : On constate qu'il y a deux cas : Le cas particulier (L=NULL) et le cas général (L ≠ NULL).

➔ Le cas particulier : C'est la même opération que celle de l'ajout en tête de liste.

➔ Le cas général : Il y a une seule différence par rapport à l'ajout en tête de liste. Au lieu de retourner nouveaunode, on retourne L car elle représente toujours la tête de la liste L.

- Suppression en tête de liste.

```
Liste suppressionEnTete(liste L)
{
  If L == NULL            /* Liste vide*/
     Return NULL ;
  Else
      If L -> suiv == L   /* Liste singleton*/
      {
        Free(L), L=NULL;
        Return NULL;
       }
      Else                /* Liste contenant plus d'un noeud*/
        {
           Node* Ptete = L; /*Pour pointer tête de liste*/
           Node* Pt = L;
           While Pt -> suiv != L
                  Pt = Pt -> suiv;
            Pt -> suiv = L -> suiv;
            Free(Ptete), Ptete = NULL;
            Return L -> suiv;
        }
     }
```

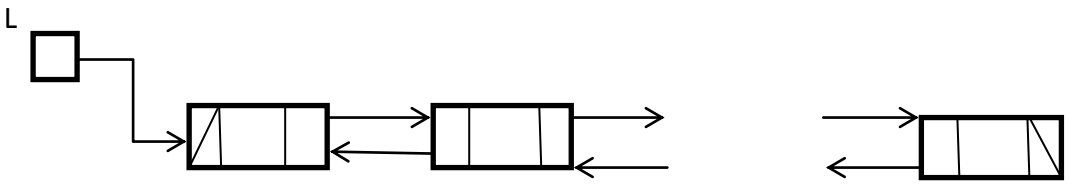
NB: On Remarque qu'il y ait trois cas possibles. Liste vide, liste singleton et liste contenant plus d'un nœud.

- Suppression en fin de liste.

```
Liste suppressionEnFin(liste L)
{
 If L = = NULL
   Return NULL ;
 Else
    If L -> suiv == L
    {
       Free(L), L = NULL;
       Return NULL;
    }
    Else
    {
       Node* Pt = L;
       Node* Ptfin = L;
       While Ptfin -> suiv != L
         {
            Pt = Ptfin;
            Ptfin = Ptfin -> suiv;
         }
       Pt -> suiv = L;
       Free(Ptfin), Ptfin = NULL;
       Return L;
    }
}
```

NB: On Remarque qu'il y a trois cas possibles, les deux premiers sont les mêmes que ceux de la suppression en tête de liste quant au troisième cas, il nous faut deux pointeurs, l'un pour pointer le dernier nœud (Ptfin) et l'autre pour pointer l'avant dernier nœud (Pt).

2. Liste chaînée linéaire bidirectionnelle.



Cette liste est constituée de nœuds de trois champs : Le premier pour l'adresse du nœud précédant (prec), le deuxième pour l'information (info) et le troisième pour l'adresse du nœud suivant (suiv).

prec info suiv

Node

- On va garder la même implémentation sauf que pour le moment le nœud est constitué de trois champs.

```
#includ <stdlib.h>
Typedef  struct node node ;
Struct node {
                Struct node *prec;
                Item info;
                Struct node *suiv;
              }
Typedef node* liste;
```

Les trois déclarations vues auparavant sont valables :

- Liste L = NULL ;
- Node* L = NULL ;
- Struct node L = NULL ;

- Quelques opérations de base.

  - Le parcours d'une telle liste peut se faire dans les deux sens, en utilisant l'adresse du nœud suivant (suiv) ou à l'aide de l'adresse du nœud précédant (prec).

- Ajout en tête de liste.

```
Liste AjoutEnTete(liste L, item valeur)
{
   Node* nouveaunode = malloc(sizeof (node)) ;
   Nouveaunode -> info = valeur ;
   Nouveaunode -> prec = NULL ;
   Nouveaunode -> sui = L ;
   If L != NULL
      L -> prec  = nouveaunode ;
   Return nouveaunode ;
}
```

- Ajout en fin de liste.

```
Liste AjoutEnFin(liste L, item valeur)
{
   Node* nouveaunode = malloc(sizeof (node)) ;
   Nouveaunode -> info = valeur ;
   Nouveaunode -> suiv = NULL ;
   If L == NULL
    {
      Nouveaunode -> prec = NULL ;  /* ou bien L*/
      Return nouveaunode ;
     }
    Else
      {
        Node* Pt = L ;
        While Pt -> suiv != NULL
              Pt = Pt -> suiv ;
        Nouveaunode -> prec  = Pt ;
        Pt -> suiv  =  nouveaunode ;
        Return  L ;
       }
    }
```

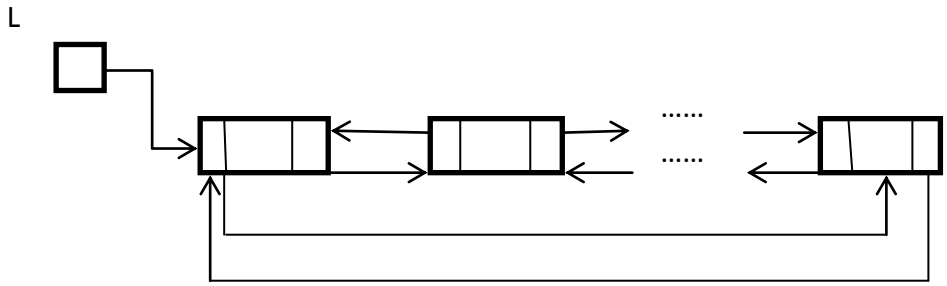
- Suppression en tête de liste.

```
Liste suppressionEnTete(liste L)
{
  If  L == NULL
     Return NULL ;
  Else
       If  L -> suiv == NULL      /* Liste singleton*/
        {
         Free(L), L = NULL;
         Return NULL;
        }
        Else       /* Liste contient plus d'un noeud*/
            {
              Node* Pt = L -> suiv;
              Pt -> prec = NULL;
              Free(L), L = NULL;
              Return Pt;
            }
   }
```

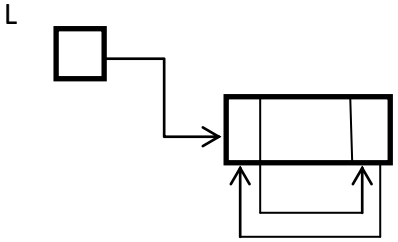
- Suppression en fin de liste.

```
Liste suppressionEnFin(liste L)
{
    If  L == NULL
      Return NULL ;
    Else
      If  L -> suiv == NULL      /* Liste singleton*/
       {
         Free(L), L = NULL;
         Return NULL;
       }
      Else
         {
           Node* PTfin = L ;
           While Ptfin -> suiv != NULL
                 Ptfin = Ptfin -> suiv ;
           Node* Pt = Ptfin ;
           Pt -> suiv = NULL ;
           Free(Ptfin), Ptfin = NULL ;
           Return  L ;
         }
    }
```

3. Liste chaînée circulaire bidirectionnelle.



Liste vide (Le seul cas où on parle de NULL)



Liste singleton

- Le parcours d'une telle liste est le même que celui de la liste linéaire bidirectionnelle sauf que :
    - Le suivant du dernier nœud est le premier nœud (L).
    - Le précédant du premier nœud est le dernier nœud.

```
Void Parcoursliste(liste L)

{

  If L == NULL

    {

      Node* Pt = L ;

      While Pt -> suiv != L

            Pt = Pt -> suiv ;

    }

}
```

NB : On peut parcourir la liste dans l'autre sens grâce à la séquence suivante.

```
While Pt -> prec != L

    Pt = Pt -> prec ;
```

- Ajout en tête de liste.

```
Liste AjoutEnTete(liste L, item valeur)
{
  Node* nouveaunode = malloc(sizeof (node)) ;
  Noveaunode -> info = valeur ;
  If L == NULL
   {
     Nouveaunode -> suiv = nouveaunode ;
     Nouveaunode -> prec = nouveaunode ;
   }
  Else
    {
       Node* Pt = L -> prec ;
       Nouveaunode -> suiv = L ;
       Nouveaunode -> prec = Pt ;   /* ou L -> prec*/
       Pt -> suiv = nouveaunode ;
       L -> prec = nouveaunode ;
     }
   Return nouveaunode ;
 }
```

- Ajout en fin de liste.

```
Liste AjoutEnFin(liste L, item valeur)
{
   Node* nouveaunode = malloc(sizeof (node)) ;
   Noveaunode -> info = valeur ;
   If L == NULL
    {
      Nouveaunode -> suiv = nouveaunode ;
      Nouveaunode -> prec = nouveaunode ;
      Return nouveaunode ;
    }
   Else
     {
        Node* Pt = L -> prec ;
        Nouveaunode -> suiv = L ;
        Nouveaunode -> prec = Pt ;   /* ou L -> prec*/
        Pt -> suiv = nouveaunode ;
        L -> prec = nouveaunode ;
        Return L ;
      }
 }
```

NB : On remarque les deux ajout (en tête et en fin de liste) sont quasiment les mêmes, sauf pour le cas de la liste non vide au lieu de retourner nouveaunode, on retourne la liste L !

- Suppression en tête de liste.

```
Liste suppressionEnTete(liste L)
{
   If L ==NULL
     Return NULL ;
   Else
     {
        Node* Ptsuivant = L -> suiv;
        Node* Ptprecedant = L -> prec;
        Ptprecedant -> suiv = Ptsuivant;
        Ptsuivant -> prec  =  Ptprecedant;
        Free(L), L = NULL;
        Return ptsuivant;
      }
   }
```

- Suppression en fin de liste.

```
Liste SuppressionEnFin(liste L)
{
  If  L == NULL
    Return NULL ;
  Else
      If  L -> suiv = L              /*L singleton*/
       {
        Free(L), L = NULL;
        Return NULL;
        }
       Else                          /*L contient plus d'un noeud*/
           {
              Node* Ptfin = L -> prec;
              Node* PtAfin = Ptfin -> prec;
              PtAfin -> suiv = L;
              L -> prec = PtAfin;
              Free(Ptfin), Ptfin = NULL;
              Return L;
            }
   }
```